Programming Languages (CS302 2007S)

On Understanding Types, Data Abstraction, and Polymorphism

Comments on:

Cardelli, Luca & Wegner, Peter. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* **17**(4), December 1985.

"Static typing is a useful property, but the requirement that all variables and expressions are bound to a type at compile time is sometimes too restrictive." Can you give an example of static typing being too restrictive?

I think the reading gives some implicit examples (and perhaps at least one explicit example). We'll go over it in class.

The section on polymorphism was very clear and it was nice to solidify what we learned earlier in the semester.

Have there been any papers written or languages created as result of a negative reaction to strongly typed languages?

Boy, I'd say that Lisp and Scheme are not strongly typed, and most modern system scripting langauges (Tcl, Ruby, Perl) are at least partial reactions to strong typing.

The reading for tomorrow was interminably long. What's worse, I got caught up in the first couple of sections. I'm unclear as to the connection between types and polymorphism. I had always thought of polymorphism in the context of OO languages. However, according to this paper, Java uses mostly ad-hoc polymorphism (overloading, coercion) and some inclusive polymorphism through inheritance. On the other hand it seems like LISP, which is weakly typed, is more polymorphic because it allows a function to take different kinds of input and returning something even if it doesn't make sense. Are strong typing techniques, then, an obstacle to true polymorphism?

I would not say that Java mostly uses ad-hoc polymorphism. We write a lot of procedures that take an interface as a parameter, which is, from my perspective, a kind of parameteric polmorphism. Java also supports inclusion polymorphism, which we've certainly used. All of the sorting routines you've written in Java use one kind of true polymorphism.

And there's not a direct correspondence between "more strongly typed" and "more polymorphic". There are certainly strongly typed systems that are also highly polymorphic. They are, in some sense, orthogonal concerns.

This reading was quite long and relatively difficult to follow. I however got a better understanding of types and polymorphism.

What's the difference between class parametrization and polymorphism? Is there a formal definition for parametrization?

Class paremeterization is a form of polymorphism. We are saying, in effect, that one (or more) fields of the class can be one of a number of types. What do you mean by a formal defintiion? Certainly, we can define them in some formal systems, but I don't think you'd want to read it

What method of OO code reuse (Polymorphism or inheritance) is preferred?

Different problems benefit from different kinds of reuse. However, it is my feeling that inheritance would be much less useful without some form of polymorphism.

Strongly typed languages do not appeal to hackers because they impose too many limitations on the programmers. Typed languages are therefore aimed at industry folks who do not trust the intelligence of their programmers.

Boy, that comment is a little extreme.

After discussing "Sets" in the previous class, we know that Sets in set theory is a type. The author tells us that "Untyped" means that there is only one type (pg. 472) and that monomorphic also means that there is only one type(pg 475). I am not sure I understand the difference between these terms.

No. "Untyped" means that there is only one type, so every value has the same type. "Monomorphic" means that each value has only one type, but different values may have different types.

I admit I was not able to go through the reading as thoroughly as I normally would because of its length.

I did warn you that the reading was very long and that you did not need to read all of it.

I did not find the syntax in the examples very intuitive. The authors explain that for the examples, the 'Fun' language is more suitable than using other 'production' programming languages. Is this because this piece was written in 1985, or do you think that Fun is still the most suitable language for the examples?

I'll admit to mixed views. Some folks very happily use strongly typed languages. Some folks much prefer weakly typed languages. Of modern languages, Haskell probably comes closest to Fun, and I expect it draws some ideas from Fun.

Also, I did not fully understand what existential quantification is because I did not understand the examples very well. Have we seen existential quantification before in class? Please can you explain more about existential quantification and universal quantification?

You haven't seen existential quantification in our discussion of types, but you've certainly seen it in CSC201.

Why are supertypes viewed as parametric types whose parameter is omitted by the user (i.e. in OOP)? Further, I don't understand why for the understanding the similarity between parametric types and supertypes, it is useful to introduce a notation in which supertype parameters must be explicitly supplied in specializing a supertype to a subtype?

Cardelli and Wegner's seem to pretty clearly favor strongly typed languages, and they present monomorphism vs. polymorphism as the only two options for a language. In Automata, the way we look at recursion theory and Turing machines states that anything can be encoded as a string or a number (whichever one we want to work with), including the machines themselves. This sort of encourages polymorphism not by having subtypes, but by making every thing the same type and making it not matter how we treat the data. This approach doesn't seem to jive with some of the ideas in this article. I realize this is sort of apples and oranges, but I'm curious how it can tie back into what we've read here.

I think Cardelli and Wegner address this issue fairly well in their discussion of untyped languages.